ICP localization

- **Iterative closest point (ICP)** is an algorithm employed to minimize the difference between two clouds of points, and widely used in localization of robot. For more details, check this.

- We can localize our vehicle’s pose by ICP algorithm given the map and the scan.

- Use PCL Library to perform ICP algorithm.

Map

- We’ll provide you a point cloud map in **.pcd** format, please put it in your package. And you can locate its path by using ***ros/package.h***.

- You will use map cloud as target cloud in ICP algorithm.

- Please have a look at [1] , [2] to learn how to load point cloud map.

Lidar point cloud

- In the challenge bag, it contains the lidar message in this type: (sensor_msgs/PointCloud2)

- Please subscribe this message/scan to perform ICP algorithm.

- Remember you need to convert point cloud from **ROSMsg** format to **PCL pointcloud** format. For implementation details, see [1].

- As you are trying to publish a point cloud, you can convert **PCL point cloud** to **ROSMsg** format first, or publish it directly. Decide by yourself.

ICP algorithm

- In PCL library, it provides lots of function for you to use, including ICP algorithm.
- Basically, ICP Localization can be decomposed into continuous-time scan matching.
- By doing scan matching, we can estimate the **transform** between two scan.
- To localize where the car is, just perform scan matching between current scan and map.

Hint

- Notice that each time you perform ICP scan matching, you may give a initial aguess (You can set it by algorithm or manually adjustment). For instance,
  icp.align(*cloud, initial_guess) // initial_guess is type of Eigen::Matrix4f
- Preprocessing point cloud before performing ICP algorithm may be useful. You can carefully use some filters (passthrough, voxel grid,...etc.) to downsample original point cloud to speed up the ICP computation and observe that if the performance become better or worse.

Hint

- Since it takes time to estimate new transform, you may need to set rosbag play rate when running the node.

- To ensure your bag time and ROS time are the same, you may need to type the following command to play bag:

  ```
  $ rosparam set use_sim_time true
  $ rosbag play -r 0.3 {challenge}.bag --clock
  ```

GPS

- GPS plays an important role in the self-driving car. With GPS data, it is able to know where we are quickly. However, in most cases the accuracy is not good enough. That’s why we only use GPS data as the **initial guess**.

- For the test dataset(NCTU campus). You can access the data by subscribing topic **/fix**.

GPS

- In the bags, it contains the GPS message:
  /fix (sensor_msgs/NavSatFix)

  You can subscribe this topic to get the longitude, latitude and altitude.

- For the GPS data details, check [1].

GPS

Here, we provide you an approach to transform the GPS data:

- GeographicLib
  which is a library for performing conversions between geographic, UTM, UPS, MGRS, geocentric, and local cartesian coordinates. You can use this library to compute the x-y coordinate according to a given map origin.

- To install the **GeographicLib** library, you can follow the instruction from this link.

GPS

- We provide a simple GPS transformation package as your reference.

- The map origin point for GPS transformation to local cartesian:
  - **lat_origin** = 24.7855644226
  - **lon_origin** = 120.997009277
  - **alt_origin** = 127.651

- You can to use other library for GPS data transformation or derive the equations by yourself as well.

tf

- ROS provides a useful package tf to maintain the relationship between different coordinate frame.

- Please write a **tf** to define the relationship between map and scan(car), your **frame_id** would be **/map** and **/scan** respectively. For more implementation details, see this.

- After you successfully publish the tf, it would be able to see the current scan and map in the same frame.

Resulting

- Your result may looks like: https://youtu.be/bN-I0YN81Ac(test in NCTU campus)